Introduction to Standard C++

Lecture 03: Class Hierarchies and Dynamic Polymorphism

Massimiliano Culpo[1]

[1]CINECA - SuperComputing Applications and Innovation Department

07.04.2014

Table of contents - Language rules

Table of contents - Language rules

Necessity for class specialization: a classic example

Consider a program dealing with people employed by a firm:

```
class Employee {
  string name_,surname_;
  Date   hiring_date_;
  /* ... */
};
```

Consider further the necessity of representing a manager:

```
class Manager {
  Employee record_;
  set<Employee*> group_;
  /* ... */
}; // Use composition as a first guess
```

Necessity for class specialization: a classic example

```
/* Prints name,surname and hiring date */
void printStatus(const Employee& emp);

/* The previous function should
   work with Manager objects */
class Manager : public Employee {
  set<Employee*> group_;
  /* ... */
};
```

1. From a software design perspective a **Manager** is an **Employee**
2. This kind of relationship is expressed in `C++` through inheritance
3. **Employee** is referred to as the base class
4. **Manager** is called the derived class

Table of contents - Language rules

Derived classes (§10)

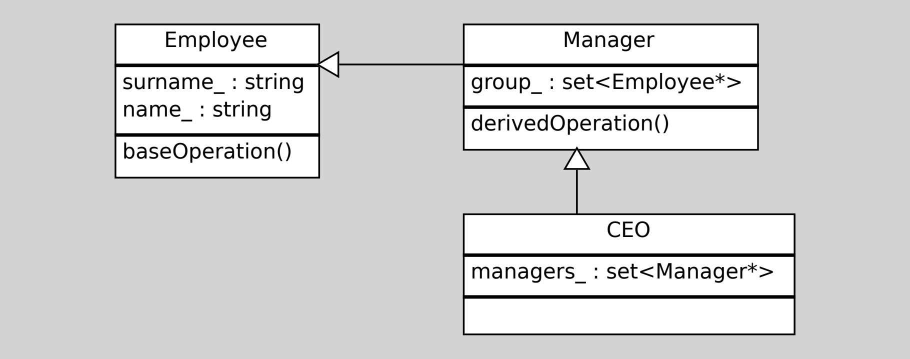
Consider the following definition:

```
class Manager : public Employee {
  /* ... */
};
```

1. **Employee** is a direct base class of **Manager**
2. In general, **A** is a base class of **B** if:
    - it is a direct base class of **B**
    - it is a direct base class of one of **B** base classes
3. Unless redeclared in the derived class, members of a base class are also considered to be members of the derived class
4. The base class members are said to be inherited by the derived class
5. Inherited members can be referred to in expressions, unless their names are hidden or ambiguous

Derived classes (§10)

A class hierarchy can be represented by a directed acyclic graph:



1. An arrow means “directly derived from”
2. A graph of this kind is often referred to as subobject lattice

Derived classes (§10)

An object may have multiple levels of inheritance:

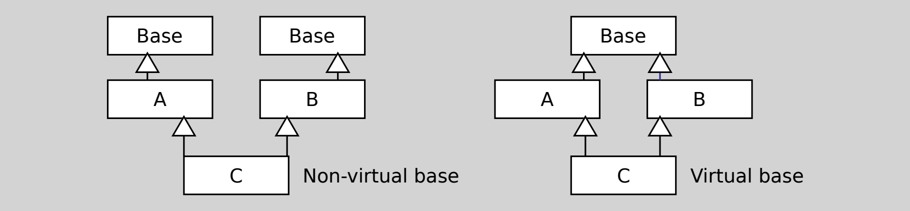
```
struct Base {
  int a, b, c;
};
struct DerivedL1 : Base {
  int b;
};
struct DerivedL2 : DerivedL1 {
  int c;
};
```

1. In this example, **Base** is:
   - a direct base class of **DerivedL1**
   - an indirect base class of **DerivedL2**

Multiple base classes (§10.1): subobject lattices

```
class X { };
class Y : public X, public X {
}; // ill-formed
struct L {
  int next;
};
class A : public L { };
class B : public L { };
class C : public A, public B {
  void f();
}; // well-formed
class D : public A, public L {
  void f();
}; // well-formed
```

Multiple base classes (§10.1): virtual base classes



1. A base class specifier that:
   - does not contain the keyword **virtual**, specifies a non-virtual base class
   - contains the keyword **virtual**, specifies a virtual base class
2. For each distinct occurrence of a non-virtual base class:
   - the derived object contains a distinct base class subobject of that type
   - explicit qualification can be used to specify which subobject is meant

Q: Can you draw the subobject lattice of A, B and C?

```
class V{};
class L : public V{};
class M : public virtual V{};
class N : public virtual V{};
class O : public M {};

class A : public M, public N {};
class B : public L, public M , public N {};
class C : public L, public N , public O {};
```

Member name lookup (§10.2)

1. Member name lookup:
   - determines the meaning of a name in a class scope
   - can result in an ambiguity, in which case the program is ill-formed
2. The scope in which name lookup begins is:

non-qualified expression	class scope of **this**
qualified expression	scope of the nested name specifier

3. The lookup-set for a name **f** in class **C** consists of:

declaration set	a set of members named **f**
subobject set	a set of subobjects where declarations were found

Member name lookup (§10.2)

```
struct A { int f(); };
struct B { int f(); };
struct C : A, B {
  int f() { return A::f() + B::f(); }
};

struct V {int v;};
struct A {int a; static int s; enum {e};};
struct B : A, virtual V {};
struct C : A, virtual V {};
struct D : B, C {};
D* pd; // Are pd->{v, s, e, a} ambiguous?
```

1. Ambiguities can often be resolved by qualifying a name
2. A static member, a nested type or an enumerator defined in a base class can be found unambiguously

Q: Which names are ambiguous? (§10.2)

```
struct V { int f(); int x; };
struct W { int g(); int y; };

struct B : virtual V, W {
  int f(); int x;
  int g(); int y;
};
struct C : virtual V, W { };
struct D : B, C { void glorp(); };

void D::glorp() {
  x++; // ??
  f(); // ??
  y++; // ??
  g(); // ??
}
```

Q: Which conversion triggers an ambiguous name look-up?

```
struct V { };
struct A { };
struct B : A, virtual V { };
struct C : A, virtual V { };
struct D : B, C { };

void g() {
  D d;

  B* pb = &d; // ??
  A* pa = &d; // ??
  V* pv = &d; // ??
}
```

Table of contents - Language rules

Virtual functions (§10.3)

1. A polymorphic class declares or inherits a virtual function
2. If a virtual member function **vf**:
    - is declared in a class **Base**
    - is declared in a class **Derived**, derived directly or indirectly from **Base**
    - **Derived**::**vf** prototype is exactly the same as **Base**::**vf**

   then **Derived**::**vf** is also virtual and it overrides **Base**::**vf**
3. A virtual member function is a final overrider unless the most derived class of which **Base** is a base declares or inherits a function that overrides it
4. In a derived class, if a virtual member function of a base class subobject has more than one final overrider the program is ill-formed
5. Even though destructors are not inherited, a destructor in a derived class overrides a base class destructor declared virtual

Example (§10.3.2): final overrider

```
struct A { virtual void f(); };

struct B : virtual A {
  virtual void f();
};

struct C : B , virtual A {
  using A::f;
};

void foo() {
  C c;
  c.f();    // calls B::f (final overrider)
  c.C::f(); // calls A::f (using−declaration)
}
```

Virtual functions (§10.3)

A virtual member function does not have to be visible to be overridden:

```
struct B {
  virtual void f();
};
struct D  : B {
  void f(int);
};
struct D2 : D {
  void f();
};
```

1. In the previous snippet:
   - the function **f**(**int**) in class **D** hides the virtual function **f**()
   - **D**::**f**(**int**) is not a virtual function
   - **f**() in class **D2** has the same name and parameter list as **B**::**f**()
2. **D2**::**f**() is a virtual function that overrides the function **B**::**f**()

Return type of overriding functions (§10.3.7)

1. The return type of a function **D::f** overriding **B::f** shall be either:
   - identical to the return type of the overridden function
   - covariant with the classes of the functions
2. The return types of the functions are covariant if:
   - both are pointers or references to classes
   - the class in the return type of **B::f**
     - is the same class as the class in the return type of **D::f**
     - is an unambiguous and accessible base class of the return type of **D::f**
   - both pointers or references have the same cv-qualification
   - the class type in the return type of **D::f** has the same cv-qualification as or less cv-qualification than the class type in the return type of **B::f**

Virtual functions (§10.3.8): example

```
class B { };
class D : private B { friend class Derived; };
struct Base {
  virtual void vf1();
  virtual void vf2();
  virtual void vf3();
  virtual B* vf4();
  virtual B* vf5();
};
struct No_good : public Base {
  D* vf4(); // error: B is inaccessible
};
struct Derived : public Base {
  void vf1(); // virtual and overrides Base::vf1()
  void vf2(int); // not virtual, hides Base::vf2()
  char vf3(); // error: invalid return type
  D* vf4();  // OK: returns pointer to derived class
};
```

Q: Is there a difference between case 1 and 2?

```
class Transaction {
public:
  Transaction();
  virtual void logTransaction() const;
};
class BuyTransaction: public Transaction {
public:
  virtual void logTransaction() const;
  BuyTransaction();
};
/* 1 */
Transaction::Transaction() { logTransaction(); }
BuyTransaction::BuyTransaction(){}
/* 2 */
Transaction::Transaction() {}
BuyTransaction::BuyTransaction(){logTransaction();}
```

Abstract classes (§10.4)

```
class point {
  /* ... */
};
class shape { // abstract class
  point center;
public:
  point where() { return center; }
  void move(point p) { center=p; draw(); }
  virtual void rotate(int) = 0; // pure virtual
  virtual void draw() = 0; // pure virtual
};
```

1. An abstract class defines an interface, therefore:
   - it can be used only as a base class of some other class
   - has at least one pure virtual function
2. Derived classes provide a variety of implementations

Abstract classes (§10.4)

```
shape x;  // error: object of abstract class
shape* p; // OK
shape f(); // error
void g(shape); // error
shape& h(shape&); // OK
```

1. An abstract class shall not be used as
   - a parameter type
   - a function return type
   - the type of an explicit conversion
2. Pointers and references to an abstract class can instead be declared
3. From these rules it follows that:
   - an abstract class can be derived from a class that is not abstract
   - a pure virtual function may override a non-pure virtual function

Abstract classes (§10.4)

A class is abstract if it inherits at least one pure virtual function:

```
class ab_circle : public shape {
  int radius;
public:
  void rotate(int) { }
  // ab_circle::draw() is a pure virtual
};

class circle : public shape {
  int radius;
public:
  void rotate(int) { }
  // a definition is required somewhere
  void draw();
};
```

Accessibility of base classes (§11.2)

The keywords **public**, **protected** and **private**:

```
class  B { };

class  D1 : private B { };
class  D2 : public B { };
class  D3 : B { }; // B private by default

struct D4 : public B { };
struct D5 : private B { };
struct D6 : B { }; // B public by default

class  D7 : protected B { };
struct D8 : protected B { };
```

may be used to set access properties of base classes

Accessibility of base classes (§11.2)

A member of a private base class:

```
struct B {  int mi; // non-static member
     static int si; // static member };
class D  : private B {};
class DD : public  D { void f(); };
void DD::f() {
  mi = 3; // error: mi is private in D
  si = 3; // error: si is private in D
  ::B b;
  b.mi = 3; // OK ( b.mi is different from this->mi)
  b.si = 3; // OK ( b.si is different from this->si)
  ::B::si  = 3; // OK
  ::B* bp1 = this; // error: B is a private base class
  ::B* bp2 = (::B*) this; // OK with cast
  bp2->mi  = 3; // OK: access through a pointer to B.
}
```

might be inaccessible as an inherited member name, but accessible directly

Q: Which of the following expressions are ill-formed?

```
class B { protected: int i; };
class D1 : public B {};
class D2 : public B {
  friend void fr(B*,D1*,D2*);  void mem(B*,D1*);
};
void fr(B* pb, D1* p1, D2* p2) {
 pb->i = 1; p1->i = 2; p2->i = 3; // ??
 int B::* pmi_B = &B::i; int B::* pmi_B2 = &D2::i; // ??
}
void D2::mem(B* pb, D1* p1) {
 pb->i = 1; p1->i = 2; i = 3; B::i = 4;// ??
}
void g(B* pb, D1* p1, D2* p2) {
  pb->i = 1; p1->i = 2; p2->i = 3; // ??
}
```

Access to virtual functions (§11.5)

```
struct B {
 virtual int f();
};
class D : public B {
private:
 int f();
};
void f() {
 D d;
 B* pb = &d;
 D* pd = &d;
 pb->f(); // OK: B::f() is public
 pd->f(); // error: D::f() is invoked (private)
}
```

1. The access rules for a virtual function:
    - are determined by its declaration
    - are not affected by the rules for a function that later overrides it

Derived classes: member inizialization (§12.6.2.10)

1. The initialization of a class object proceeds in the following order:

virtual base classes	initialized in the order they appear (depth-first left-to-right traversal)
direct base classes	initialized in declaration order
non-static data members	initialized in declaration order
constructor body	executed after member initialization

2. The declaration order is mandated to ensure that base and member subobjects are destroyed in the reverse order of initialization
3. Member functions (including virtual member functions) can be called for an object under construction
4. However, if these operations are performed in an initializer the result of the operation is undefined

Table of contents - Language rules

Dynamic cast (§5.2.7)

The result of the expression:

```
dynamic_cast<T>(v)
```

1. Converts the expression **v** to type **T**
2. **T** shall be a pointer or reference to a complete class type or **void** $*$
3. The **dynamic_cast** operator shall not cast away constness
4. If the type of **v** is the same as **T** the result is **v**
5. If **v** is a null pointer value, the result is the null pointer value of type **T**
6. If **T** is pointer to B and v has type pointer to D such that **B** is a base class of **D**, the result is a pointer to the unique **B** subobject of the **D** object pointed to by **v**
7. Otherwise, **v** shall be a pointer to or an lvalue of a polymorphic type

Dynamic cast (§5.2.7)

```
class A { virtual void f(); };
class B { virtual void g(); };

class D : public virtual A, private B { };

void g() {
  D d;
  B* bp = (B*) &d; // cast needed to break protection
  A* ap = &d; // public derivation, no cast needed
  D& dr = dynamic_cast<D&>(*bp); // fails
  ap = dynamic_cast<A*>(bp); // fails
  bp = dynamic_cast<B*>(ap); // fails
  ap = dynamic_cast<A*>(&d); // succeeds
  bp = dynamic_cast<B*>(&d); // ill-formed
}
```

Static cast (§5.2.9)

The result of the expression:

```
static_cast<T>(v)
```

1. Converts the expression **v** to type **T**
2. The **static_cast** operator shall not cast away constness
3. If the declaration **T t(e)** is well-formed, **e** is converted to type **T**
4. The effect of such an explicit conversion is the same as:
   - performing the declaration and initialization
   - using the temporary variable as the result of the conversion
5. An lvalue of type **B** can be cast to **D**& when:
   - **D** is a class derived from **B**
   - a valid standard conversion from pointer to **D** to pointer to **B** exists
   - **B** is not a virtual base class of **D**

Table of contents - Best practices

Table of contents - Best practices

DISALLOW WHAT YOU DON'T WANT (PART 2)

A class **Uncopyable** that barely can't be copied or assigned:

```
class Uncopyable {
protected:
  Uncopyable()  {};
  ~Uncopyable(){};
private:
  Uncopyable(const Uncopyable&);
  Uncopyable& operator=(const Uncopyable&);
};
```

can be used as a policy to propagate the behavior in derived classes:

```
class Derived : private Uncopyable {
  // This class will fail at compile-time in case
  // copy-constructor or assignment are generated
};
```

DESTRUCTOR IN POLYMORPHIC BASE CLASSES

```
class TimeKeeper {
public:
  TimeKeeper();
  virtual ~TimeKeeper();
  /* ... */
};

class AtomicClock: public TimeKeeper { /* ... */ };
class WaterClock:  public TimeKeeper { /* ... */ };
```

As many clients will access a dynamic object through a pointer or reference to its polymorphic base, the use of a virtual destructor is mandatory:

```
// Get dynamically allocated object
TimeKeeper *ptk = getTimeKeeper();
/* ... */
delete ptk;
// What if a non-virtual destructor is used here?
```

Table of contents - Best practices

MAKE INTERFACES EASY TO USE CORRECTLY

Developing effective interfaces requires that you consider the kinds of mistakes that clients might make:

```
class Date {
public:
  Date(int month, int day, int year);
};
```

At a first glance this may seem reasonable, but:

```
Date d(30, 3, 2013); // should be d(3,30,2013)
Date d(30,30, 2013); // should be d(3,30,2013)
```

These client errors may be prevented by the introduction of new types:

```
struct Day {
  explicit Day(int d) : val_(d) {}
  int val_; };
```

MAKE INTERFACES EASY TO USE CORRECTLY

The new definition of **Date** would be:

```
class Date {
public:
  Date(const Month& month,
       const Day&   day,
       const Year&  year);
};
/* ... */
Date d(30, 3,2013); // error
Date d(Day(30), Month(3), Year(2013)); // error
Date d(Month(3), Day(30), Year(2013)); // OK
```

Once the right types are in place, it is reasonable to:

- restrict the values of those types
- restrict the set of operations that are allowed on those types
- ensure a behavior that is as compatible as possible with built-in types

CLASS DESIGN IS TYPE DESIGN

Good class have natural syntax and intuitive semantics:

1. How should objects of your new type be created and destroyed?
2. How should object initialization differ from object assignment?
3. What does it mean for objects to be passed by value?
4. What are the restrictions on legal values for your new type?
5. Does your new type fit into an inheritance graph?
6. What kind of type conversions are allowed for your new type?
7. What operators and functions make sense for the new type?
8. What standard functions should be disallowed?
9. Who should have access to the members of your new type?
10. What is the "undeclared interface" of your new type?

PREFER NON-MEMBER NON-FRIEND FUNCTIONS

Object oriented principles dictate encapsulation, but usually it is misunderstood how the principle should be put into practice:

```
class WebBrowser {
public:
  void clearCache();
  void clearHistory();
  void removeCookies();
};
```

There are at least two alternatives to perform these actions together:

```
/* Member function */
void WebBrowser::clearAll() {
/* ... */
}
/* External function */
void clearBrowser(WebBrowser& browser);
```

PREFER NON-MEMBER NON-FRIEND FUNCTIONS

In C++ the best solution to this problem is the use of convenience functions with a common namespace, but defined is separate translation units:

```
/* header "webbrowser.h" */
namespace WebBrowserStuff {
  class WebBrowser { /* ... */ };
}
/* header "webbrowsercookies.h" */
namespace WebBrowserStuff {
  // Cookies related convenience functions
}
```

This approach effectively implements three basic OO principles:

1. encapsulation
2. packaging flexibility
3. functional extensibility

CONVERT ALL THE PARAMETERS OF A FUNCTION

Consider the following snippet:

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  /* ... */
};
```

The correct way to support mixed-mode arithmetic operations for a class of this kind is through non-member functions:

```
const Rational operator*(const Rational& lhs,
                         const Rational& rhs);
```

as it allows compilers to perform implicit type conversions on all arguments.

Table of contents - Best practices

PUBLIC INHERITANCE MODELS "IS-A"

The equivalence of public inheritance and "is-a" relationship sounds simple, but sometimes your intuition can mislead you:

```
class Bird {
public:
  virtual void fly() = 0;
};

class Penguin : public Bird {
  /* Can they fly?!? */
};
```

To implement an "is-a" relationship you must ensure that:

- everything that applies to base classes must also apply to derived classes

because every derived class object is a base class object.

AVOID HIDING INHERITED NAMES

Consider the following snippet:

```
class Base {
public:
  void mf3(double& in); };
class Derived: public Base {
public:
  void mf3(int& in); };
```

In this case **Derived::mf3** hides **Base::mf3**. This is never desirable in case of public inheritance. The correct way to extend the look-up set of a name is through a **using** declaration:

```
class Derived: public Base {
public:
  using Base::mf3;
  void mf3(int& in);
};
```

INHERITANCE OF INTERFACES AND IMPLEMENTATIONS

Consider the following class hierarchy:

```cpp
class Shape {
public:
  virtual void draw() const = 0;
  virtual void error(const std::string& msg);
  int objectID() const;
};
class Rectangle: public Shape { /* ... */ };
class Ellipse  : public Shape { /* ... */ };
```

There is clearly a difference in the semantic of the three declarations:

pure virtual function	derived classes inherit a function interface only
virtual function	derived classes inherit a function interface as well as a default implementation
non-virtual function	derived classes inherit a function interface as well as a mandatory implementation

COMPOSITION MODELS "HAS-A"

Composition is the relationship between types that arises when objects of one type contain objects of another type:

```
class Address { /* ... */ };
class PhoneNumber { /* ... */ };
class Person {
public:
  /* ... */
private:
  std::string name;
  Address address;
  PhoneNumber voiceNumber;
  PhoneNumber faxNumber;
};
```

It's meaning is completely different from that of public inheritance:

application domain	models an "has-a" relationship
implementation domain	models an "is implemented in terms of" relationship

USE OF PRIVATE INHERITANCE

An implementation detail can be coded using private inheritance:

```
class Widget: private Timer {
private:
  virtual void onTick() const; };
```

Private inheritance can usually be avoided resorting to private classes:

```
class Widget {
private:
  class WidgetTimer: public Timer {
  public:
    virtual void onTick() const;    };
  WidgetTimer timer; };
```

Composition is to be preferred to this approach, though it makes sense:

- when a derived class needs access to protected members
- when a derived class needs to redefine inherited virtual functions

USE OF MULTIPLE INHERITANCE

```
class IPerson { // Interface to be implemented
public:
  virtual ~Iperson(){}
  virtual std::string name() const = 0; };
class PersonInfo { // Helps in implementing an IPerson
public:
  const char * theName() const; };
class CPerson : public IPerson, private PersonInfo {
public:
 virtual std::string name() const {
   std::string name(theName);
   return name;
 } };
```

One of the most common use case of multiple inheritance is:

public inheritance from an interface

private inheritance from a class that helps with implementation

The one slide summary of the lecture

Class hierarchies and polymorphism

1. A class may inherit from multiple base classes
2. Virtual functions may be overridden to obtain a polymorphic behavior
3. Abstract classes define an interface and cannot be instantiated

Best practices

1. Destructors in polymorphic base classes must be virtual
2. Public inheritance models the is-a relationship
3. Composition or private inheritance model the has-a relationship
4. Avoid hiding inherited names, use multiple inheritance judiciously

Table of contents - Appendices

Meyers, S.
Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd ed.
Addison-Wesley Professional, 2005.

Sutter, H.
Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

Sutter, H., and Alexandrescu, A.
C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, 1 ed.
Addison-Wesley Professional, Nov. 2004.

Table of contents - Appendices

Member look-up set (§10.2): definition and calculation

The following steps define how the look-up set $S(f, C)$ is constructed:

- if **C** contains a declaration of **f**:

  declaration set contains every declaration of **f** in **C**
  subobject set contains **C**
- otherwise $S(f, C)$ is initially empty
- if **C** has direct base classes $\mathbf{B}_i$ $i = 1, \ldots, n$
  1. calculate $S(f, B_i)$ for $i = 1, \ldots, n$
  2. merge all the $S(f, B_i)$ into $S(f, C)$

Member look-up set (§10.2): definition and calculation

The following steps define the result merging process:

1. $S(f, C)$ is unchanged if:
   - each of the subobject members of $S(f, B_i)$ is a base class subobject of at least one of the subobject members of $S(f, C)$
   - $S(f, B_i)$ is empty
2. $S(f, C)$ is a copy of $S(f, B_i)$ if:
   - each of the subobject members of $S(f, C)$ is a base class subobject of at least one of the subobject members of $S(f, B_i)$
   - $S(f, C)$ is empty
3. $S(f, C)$ is ambiguous (invalid declaration set) if:
   - $S(f, B_i)$ and $S(f, C)$ differ

   An invalid declaration set is considered different from any other
4. Otherwise, the new $S(f, C)$ is a lookup set with the shared set of declarations and the union of the subobject sets

Member look-up set (§10.2.7): example

```
struct A { int x;   };// S(x,A) = {{ A::x }, { A }}
struct B { float x; };// S(x,B) = {{ B::x }, { B }}

// S(x,C) = {invalid, { A in C, B in C }}
struct C: public A, public B { };
// S(x,D) = S(x,C)
struct D: public virtual C { };
// S(x,E) = {{ E::x }, { E }}
struct E: public virtual C { char x; };
// S(x,F) = S(x,E)
struct F: public D, public E { };

int main() {
  F f;
  f.x = 0; // OK, lookup finds E::x
}
```